Efficiency evaluation of overhead control heuristics in DP-Fair multiprocessor scheduling

Yvon Trinquet, Naeem M. Shehzad, Anne-Marie Déplanche, Richard Urunuela

Efficiency evaluation of overhead control heuristics in DP-Fair multiprocessor scheduling

M.Naeem Shehzad, A.M Déplanche, Yvon Trinquet, Richard Urunuela
LUNAM Université, Université de Nantes, IRCCyN UMR CNRS 6597
L'Institut de Recherche en Communications et Cybernétique de Nantes
Nantes, France.
{Naeem.Shehzad,Anne-Marie.Deplanche,Yvon.Trinquet,Richard.Urunuela}@irccyn.ec-nantes.fr

Abstract

A number of optimal algorithms exist for scheduling of periodic taskset with implicit deadlines in real-time multiprocessor systems. However, the practical facts reveal that the optimality is achieved at the cost of excessive scheduling points, migrations and preemptions. In [20], we proposed two heuristics to control the overhead for a class of non-work conserving global scheduling algorithms that combine fluid scheduling and deadline partitioning, while guaranteeing optimality. This paper gives some detailed simulation results along with description of the system to generate the data for the simulation. The given results show the basic strength of the heuristics and validate their efficiency. [1]

1. Introduction

The significance of real-time systems can be viewed everywhere from mobile phones and automobiles upto space systems. This revolution of technology has constantly urged for an increase in the processing power. Multiprocessing addresses to this problem. Scheduling problem in real-time multiprocessor systems has got a lot of attention after emergence of multicore architecture [4, 14, 19]. The two main categories of multiprocessor scheduling algorithms in real-time systems are partitioned scheduling and global scheduling.

In different to partitioned scheduling [12, 13, 8], there is a single queue of ready tasks and a single scheduler for all the processors in global scheduling. Tasks and their jobs are allowed to migrate from processor to processor. Migration increases the schedulability which consequently improves the resource utilization. Another significant advantage of global scheduling is that the only known optimal multiprocessor scheduling algorithms for periodic tasks belong to this category.

[1] *This work has been supported by the French Agence Nationale de la Recherche through the RESPECTED project (Contract ANR-2010-SEGI-002). See http://anr-respected.laas.fr*

Though theoretically optimal, questions are raised about the practical implementation of global scheduling. The optimality is achieved at the rate of a large number of migrations, preemptions and scheduling points, the cost of which is commonly considered to be zero or negligible. But practically their effect in the system cannot be neglected specially when these occur frequently. The cost of migration on some modern multicore architectures is much lower than in past but is still a non-zero value. Thus frequent migrations and preemptions in the system lead to an increase in worst-case execution times which may result in the missing of deadlines.

Keeping in mind the overhead due to migration and preemption, the study of their effect on different aspects of processor scheduling is quite common [5, 24, 25]. Some researchers have worked in the domain of overhead control due to preemption and migration in global scheduling. [10] used the technique of delayed preemption for preemption control in non-optimal global scheduling. Aoun et al. [3] used the processor affinity technique to reduce the number of migrations in $PFair$ scheduling. Megel et al. [16] proposed a linear programming formulation and a local scheduler technique. In [20], we proposed the use of some simple heuristics to control the preemptions and migrations while still keeping the optimality, for a class of non-work conserving global algorithms. The results showed some significant improvement in the overhead.

The known optimal global scheduling algorithms can be divided into two main classes. $PFair$ [19] and deadline partitioning fair which is shortly known as DP-Fair [14]. Both categories are based on the principle of fairness but they differ on how much fairness is required.

Proportionate fair, or simply $PFair$, was presented by Baruah et al. in 1996 [19]. It uses the concept of fairness which suggests that processor share of each task is proportional to its utilization factor at any instant. For a task T_i with utilization factor $T_i.u$, time allocated at any given time t, will be either $\lfloor t * T_i.u \rfloor$ or $\lceil t * T_i.u \rceil$. The time is divided into small intervals of equal length in $PFair$ and an interval is called quantum. Scheduling of all the tasks is done at the start of each quantum. As a result, $PFair$

achieves the optimality at the cost of huge runtime overhead [21] due to frequent scheduling points, preemptions and migrations. PF, PD and PD^2 are three $PFair$ algorithms which are proven to be optimal [19]. ERFair[2] which is a work-conserving technique is a extended form of PFair.

DP-Fair combines the notion of fluid scheduling (ideal fairness) with deadline partitioning while still guarantees the optimality. All the DP-Fair strategies choose to subdivide time into slices where all the tasks have a common (local) deadline. Such common deadlines are necessary because according to Hong and Leung [11], no optimal on-line scheduler can exist for a set of jobs with two or more distinct deadlines on two or more processors. These deadlines are called boundaries and are defined by the points of task release (that coinicide with task deadlines for an implicit deadline taskset). These are abbreviated as b_0, b_1,...b_k etc. as shown in the figure 1. Distance between any two boundaries is also known as a $node$. In DP-Fair, the fairness is required to achieve only at the boundaries. It reduces the number of scheduling points compared to that of $PFair$. Scheduling in each node comprises two steps. The computation of execution time units (called *local execution time*) for each task for that node and decision for dispatching these time units among the processors. The execution time units are called local execution time units because they are assigned only for that particular interval and can be different to the execution time of the task. Local execution time units may be a discrete or a non-discrete value depending on the technique used for their computation.

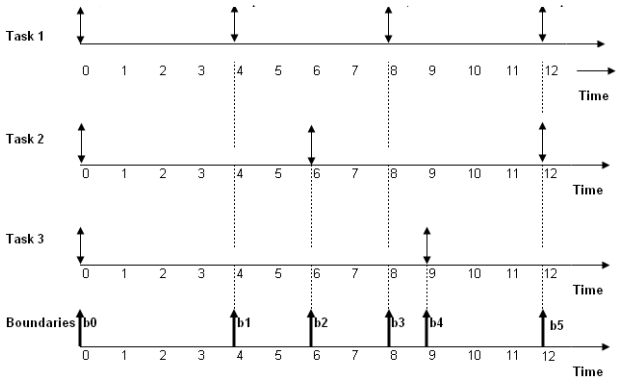

Figure 1. Definition of intervals in DP-Fair

Some of well known non-work conserving algorithms following the principle of DP-Fair are Boundary fair [26], $LLREF$ [9] and $LRE-TL$ [7]. They differ either in computation of local execution times or dispatching technique.

Boundary fair or $BFair$ is a technique proposed by Zhu et al. [26]. For a task T_i with utilization factor $T_i.u$, time allocated for any node between b_k and b_{k+1} is either $\lfloor (b_{k+1}-b_k)*T_i.u \rfloor$ or $\lceil (b_{k+1}-b_k)*T_i.u \rceil$. $BFair$ uses McNaughton's algorithm [15] for dispatching of taskset in a static way. For 100 tasks in the taskset, Zhu showed that $BFair$ has 48 % scheduling points when compared with PD [26].

In $LLREF$ [9], the execution time units of any task are directly proportional to its utilization factor at any time. The resulting value may be a non-integer which causes a practical problem during implementation (due to the hardware characteristics of processors, execution time unit numbers should be integral multiples of the highest precision timer). $LLREF$ uses a dynamic technique for dispatching of taskset. $LRE-TL$ [7] is based on the principle of $LLREF$ but defines an improved dispatching technique.

Recently, an optimal global scheduling algorithm, known as RUN, is proposed by Regnier et al. [18] which is based on a principle other than fairness. It reduces the multiprocessor problem to a series of uniprocessor problem and consequently minimizes the number of migrations and preemptions. Nelissen et al. [17] has also proposed a technique called U-EDF to reduce the number of preemptions and migrations by releasing fairness. However, the optimality is not shown for this technique. In addition, it involves non-discrete values of execution times causing some constraints during implementation.

The DP-Fair algorithm on which our work is based uses the $BFair$ [26] technique for the computation of local execution time units and the $LRE-TL$ [7] dynamic technique for the task dispatching. In [6], Cho et al. used the same algorithm. In this article, we refer it as BFair/LRE-TL.

The motivation behind this work is to elaborate the advantages of the heuristics proposed in [20] while tested in a more sophisticated way with our BFair/LRE-TL algorithm. The article discusses the whole mechanism of generating the dataset for testing the algorithm. It gives some new perspectives about the performance of these heuristics when the ratio of tasks to processor changes.

System Model. We consider that T is a set of N synchronous periodic tasks T_i where $i = 1, 2...N$ to be scheduled on M identical processors. Each task T_i has a period $T_i.p$ equal to its relative deadline $T_i.d$ (implicit deadline), an execution time $T_i.e$ and utilization factor $T_i.u$ (which is $T_i.e/T_i.p$) in range (0, 1] such that $\sum_{i=1}^{N}(T_i.u) \leq M$. Each task in such a system is invoked or released repeatedly in accordance with its period $T_i.p$. Each such invocation is called a job of the task. We assume that all the tasks are independent, i.e. they do not share any common resource and do not have any precedence with each other. The costs of migration, preemption and context switch are assumed to be already added in execution times. A processor can not execute more than one task at any given time and a single task cannot execute on more than one processor at any given time.

The organization of the rest of the paper is as follows. Section 2 discusses the DP-Fair scheduling. Section 3 briefly revises the heuristics proposed in [20]. Section 4 discusses the experimental set up along with the data generation process. Section 5 gives the experimental results

along with the conclusion in section 6.

2. Scheduling of taskset in BFair/LRE-TL

As presented earlier, in BFair/LRE-TL, the scheduling of a taskset in a node is divided into two parts, computation of local execution time units and dispatching of taskset.

2.1. Local execution time computation

We used $BFair$ [26] algorithm for the computation of task local execution times. This is realized by allocation of some mandatory units which enable each task to achieve the lower limit of fairness and then unallocated units of time over the node are are distributed as optional units according to some priority rules. Any task cannot have more than one optional unit. The resulting local execution time $T_i.l$ is sum of mandatory units and optional unit and it is always an integer value. Such a computation guarantees :

- Each local execution time is less than or equal to length of the node, i.e. $T_i.l \leq (b_{k+1} - b_k) \ \forall \ \ 0 < i \leq N$
- The sum of local execution time units is less than or equal to total capacity of the processors, i.e. $\sum_{i=1}^{N}(T_i.l) \leq M(b_{k+1} - b_k)$
- At each b_k, and for each task, the distance between its ideal fair execution and real execution is strictly less than one, ensuring that deadline will be met.

2.2. Task dispatching

We are inspired by the principle of $LRE - TL$ [7] for designing of our dispatching technique. It uses the notion of zero local laxity for tasks. A task is said to have zero local laxity if its remaining local execution time to be consumed becomes equal to the remaining time of the node. The following general rules are applied while dispatching taskset between two boundaries b_k and b_{k+1}:

- At most M tasks can be executed at any given time.
- A task with zero local laxity is given maximum priority. It must be executed immediately, otherwise it will miss its local deadline.
- A task with zero local remaining execution time is preempted (due to non-work conserving behavior).
- No processor remains idle if there is a ready task with non-zero local remaining execution time.

Both of our overhead control heuristics are related to the dispatching technique.

3. Overhead control

Generally, the theory of global scheduling only requires execution of tasks so that all tasks meet their deadline without specifying any particular task executing on any particular processor. This is the reason why most of the scheduling algorithms give no explicit prescription about assigning the tasks to the processors. Thus, after the computation of local execution times and establishment of scheduling rules, some complementary dispatching technique can be designed to reduce the overhead due to preemption and migration. In [20], we added some simple heuristics in the dispatching technique to make it more efficient in terms of overhead. The heuristic 1 is related to the task to processor assignment criterion. Heuristic 2 explores the order in which at most M tasks are selected for execution.

3.1. Heuristic 1

Normally, the assignment of tasks for execution is made to any available processors without considering their previous histories. Contrarily, heuristic 1 allows the task to keep the record of the processor on which it was executed last time and as a result an affinity relation exists between task and processor. Taking this relation into account, heuristic 1 tries to assign a newly running task to the processor on which it was scheduled the last time. This heuristic works at primary scheduling points, i.e. the ones that coincides with boundaries time (or start of a node) as well as at secondary scheduling events that may occur inside a node. The algorithm is given below. The computational complexity of heuristic 1 is $O(M)$.

Algorithm Heuristic 1
Suppose:

- t is the time at which algorithm is called
- H_B is the list of tasks that have been already selected for running after t. Maximum size of H_B is M
- $T.getLastProc()$ returns the processor on which task T was executed last time
- P is an object that represents a processor

```
1.  for(each task T of H_B not running before t)
2.        P = T.getLastProc() ;
3.        if (P is idle)
4.              assign T to P;
5.        else
6.              assign T to any idle processor;
7.  end for
```

3.2. Heuristic 2

At the start of each node, heuristic 2 attempts to control the preemptions. According to this technique, the task executing on a processor just before the scheduling is given priority to re-execute provided it is still ready. By continuing such executions, some unnecessary preemptions are avoided. The algorithm is given below. The computational complexity of heuristic 2 is $O(M)$.

Algorithm Heuristic 2
Suppose

- t is the time at which algorithm is called
- H_B is the list of tasks that were running before t and that may be updated by this algorithm. And at the end, it contains the tasks that have to run after t
- $ReadyList$ is the list containing unsorted ready tasks

1. for (each task T of H_B)
2. if($T \epsilon ReadyList$)
3. $ReadyList$.remove(T);
4. else
5. T.preempt();
6. H_B.remove(T);
7. end for
8. while (H_B.size() < M && $ReadyList.size() != 0$)
9. T = $ReadyList$.getfirst();
10. $H_B.add(T)$;
11. endwhile

4. Experimentation

A series of simulation based experimental studies was performed to find the effect of using the overhead control heuristics with BFair/LRE-TL and to discover out some relevant features. In the first place, we added heuristic 1, then heuristic 2 and finally both the heuristics with the BFair/LRE-TL to find their individual and combined effects on the migration and preemption compared to the original version of BFair/LRE-TL algorithm. When we use both the heuristics, we call the resulting algorithm as a hybrid algorithm. The experimental test bed is shown in figure 2 and explained hereafter.

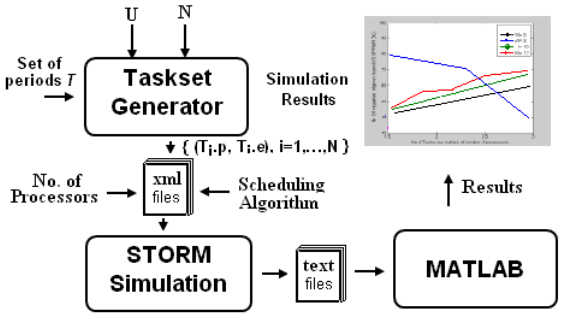

Figure 2. Experimental test bed

4.1. Taskset Generator

The functional diagram of the taskset generator is shown in figure 3. Taking total utilization factor U and number of tasks N as inputs, it gives N couples of $T_i.e$ and $T_i.p$ such that $\sum_{i=1}^{N}(\frac{T_i.e}{T_i.p}) = U$. The time periods $T_i.p$ are chosen from a set of time periods in a round robin way. Utilization of limited values of task periods helps to limit the hyper period of the taskset and thus bounds reasonably the simulation interval. Moreover, the utilization of different period sets gives rise to various distributions of node length and frequency. We used Roger Stafford's randfixedsum algorithm [22] at the heart of the generator. Stafford's algorithm efficiently generates N values between a and b such that their sum gives a constant value. The Matlab implementation of the algorithm is publicly available with all necessary documentation [22]. In our case, Stafford's algorithm takes the total number of tasks N, their total utilization factor U and limits 0, 1 as inputs and gives utilization factors $T_i.u_s$of N tasks as output. The values of of $T_i.e$ obtained from $T_i.u_s$ may not be a discrete value. Therefore, the following algorithm is used to obtain the discrete couples of $T_i.e$ and $T_i.p$ from $T_i.u_s$, while keeping the value of total utilization factors produced by Stafford's algorithm.

Algorithm for task parameters computation

Suppose
T = $\{P_j, j = 0, k-1\}$ = Set of periods
$\Delta\%$= Percentage error
Initial conditions
$\Delta_0 = 0$;
$\Delta\% = 0$

1. for ($i = 1...N$)
2. $T_i.u' = min\{(T_i.u_s + \Delta_{i-1}), 1\}$
3. $T_i.p = P_{(i\,\%\,k)}$
4. $T_i.e = max\{\lfloor T_i.p * T_i.u' \rfloor, 1\}$
5. $\Delta_i = T_i.u_s - \frac{T_i.e}{T_i.p}$
6. $\Delta\% = \Delta\% + \frac{|\Delta_i|}{T_i.u_s}$

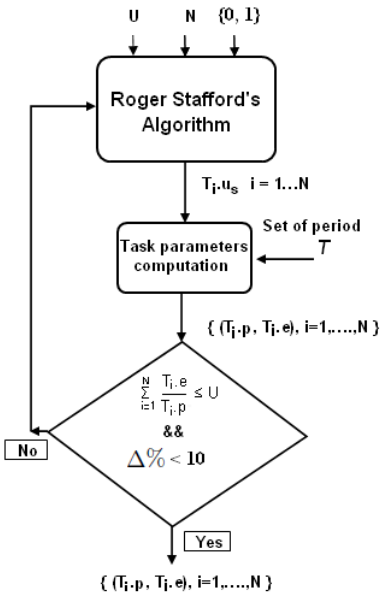

Figure 3. Dataset generator

7. *end for*
8. $\Delta\% = \frac{\Delta\%}{N}$

The taskset is discarded if the sum of utilization factor finally calculated is more than U or average error of the taskset is not less than 10 %. This value is a good compromise between the initial Stafford's distribution and a moderate time to generate a large number of configurations.

The task utilization factors produced by our generator lie uniformly between 0 and 1 when tested for U = 0.5N. At U = 0.25N, the majority of the generated configurations include light tasks. In the same way, at U = 0.75N, most of the tasks in the configurations are heavy tasks.

4.2. Experimental conditions

We have presented the results of only one set of periods due to space limitation. We used $\boldsymbol{T}$ ={30, 36, 40, 45, 50} for the experiments. It gives rise to the scheduling intervals of variable lengths between 2 and 30 with a slightly high proportion of interval of length 10. The hyper period is 1800.

1. The process of experiments was conducted at variable total utilization factor, i.e. U=M, U=0.75M and U= 0.5M.

2. For each value of total utilization factor, experiments were performed with varying number of processors including 4, 6, 8, 10 and 12.

3. For each number of processor value, four different values of number of tasks were used, i.e. N=1.5M, N=2M, N=2.5M and N=3M.

4. For each value of N, 30 configurations were generated and were tested for four algorithms, *i.e.* BFair/LRE-TL, BFair/LRE-TL with heuristic 1, BFair/LRE-TL with heuristic 2 and BFair/LRE-TL with both heuristics known as hybrid.

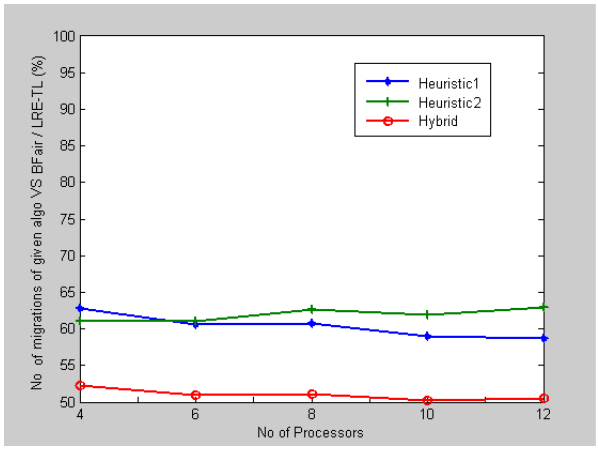

Figure 4. Migration control at U=M

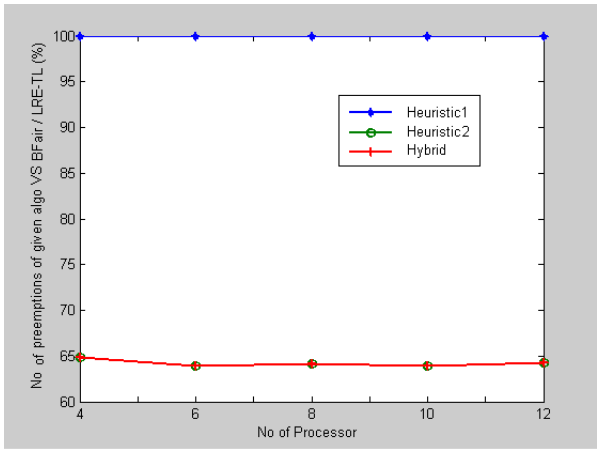

Figure 5. Preemption control at U=M

4.3. Simulator

We used STORM [1, 23] as the simulation tool. STORM stands for "Simulation TOol for Real time Multiprocessor scheduling". STORM is a freeware software tool developed in our research team. It is able to simulate the behavior of predefined or user defined real-time multiprocessor schedulers and to evaluate their performance by computing specified metrices on the schedules they construct. It has the ability to show the execution of a given set of tasks over a multiprocessor architecture while taking into account the requirements of both taskset and hardware system. It takes all the input information about the taskset, processors and the scheduling algorithm in the form of an XML file. STORM enables to incorporate an observer of the simulation as a separate program. It counts the number of migrations and preemptions without affecting the scheduling process. It stores the results in text files.

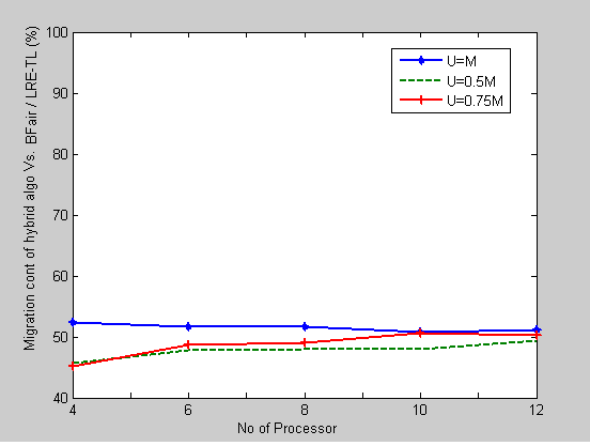

Figure 6. Migration control of hybrid algorithm with variable utilization factor

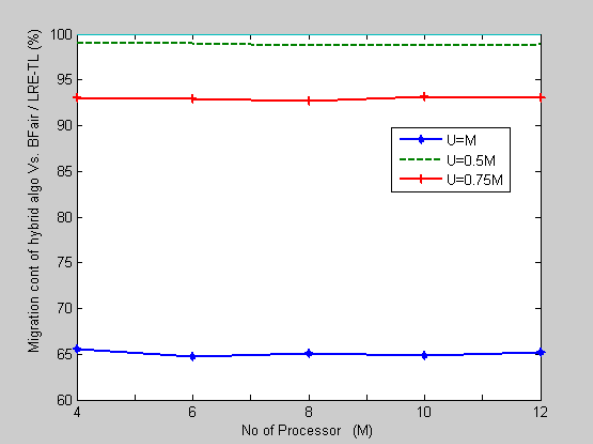

Figure 7. Preemption control of hybrid algorithm with variable utilization factor

5. Results

The results give the improvement expressed as a percentage of specific algorithm relative to the basic BFair/LRE-TL one for the number of preemptions and migrations as well. Different perspectives of the obtained results are given in the following.

5.1. Overhead control at U=M

The results are given in the graphical form in figures 4 and 5. Each point on the graph represents an average result of experiments on 120 tasksets with a variable number of task to processor ratio. The configurations include taskset with different utilization factors, light weight as well as heavy weight tasks as defined in point 3 of experimental conditions.

The results show that migrations with heuristic 1 are about 60% of BFair/LRE-TL algorithm while it shows not any effect on the number of preemptions. The preemptions with heuristic 2 are approximately 65 % of the preemptions of BFair/LRE-TL. The migrations with heuristic 2 are also in the same range. Heuristic 2 basically controls the preemptions but it reduces the migrations as well because tasks which avoid preemption by heuristic 2 may have been migrated to different processors otherwise. Hybrid algorithm keeps the advantages of both heuristic 1 and heuristic 2 and shows better results upto 50 % for migration.

5.2. Overhead control at variable total utilization factor

The graphs in the figures 6 and 7 show the results where the performance of the only hybrid algorithm is tested with different total utilization factors. Each point in the graph shows an average result of experiments on 120 tasksets with a variable number of task to processor ratio. The hybrid algorithm improves the migration control with a reduction in total utilization factor. The preemption control of the hybrid algorithm works notably better for high total utilization factor. As mentioned earlier, the preemption control heuristic works at primary scheduling points. With comparatively low utilization factor, tasks are preempted before their completion (due to the non-work conserving behavior), letting the time idle at the end of the node and making the heuristic 2 inoperative. Smaller is the total utilization factor, lower are the chances of working of heuristic 2 and lower is the preemption control.

5.3. Overhead control at a variable ratio of task to processor

The results in figures 8 to 11 show the percentage improvement in migration and preemption control of hybrid algorithm while varying the number of the task to processor ratio. Each point in the graph shows an average result of experiments on 30 tasksets. The graphs in figures 8 and 9 show the migration control and figures 10 and 11 show the preemption control of the hybrid algorithm. The experiments were done at $U = 0.75M$ and $U = M$. Number of tasks N varies between 6 and 36 according to variation in the number of processors.

The migration control works better at lower ratio of task to processor than at higher values. This trend is more significant at $U = 0.75M$ than $U = M$. The preemption control also performs better at lower values of task to processor ratio, when $U = 0.75M$. When $U = M$, the hybrid algorithm shows the best preemption control at $N = 2M$ as shown in the figure 11.

We know that preemption control ability of the hybrid algorithm due to heuristic 2 is relatively weaker at lower value of total utilization factor. Also as the ratio of number of task to processor increases, the number of light tasks in the taskset increases. Both of these factors reduce the chances of tasks to continue upto next node. It leaves the idle time units at the end of the node which prohibits the work of heuristic 2 which controls both migrations and preemptions. This is the reason of deterioration of pre-

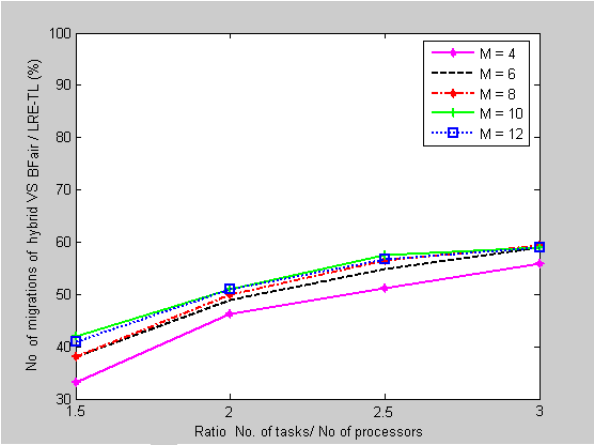

Figure 8. Migration control in variation with number of tasks at U=0.75M

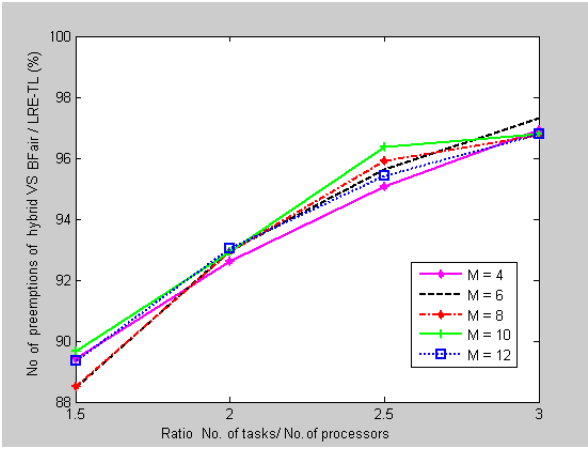

Figure 10. Preemption control in variation with number of tasks at U=0.75M

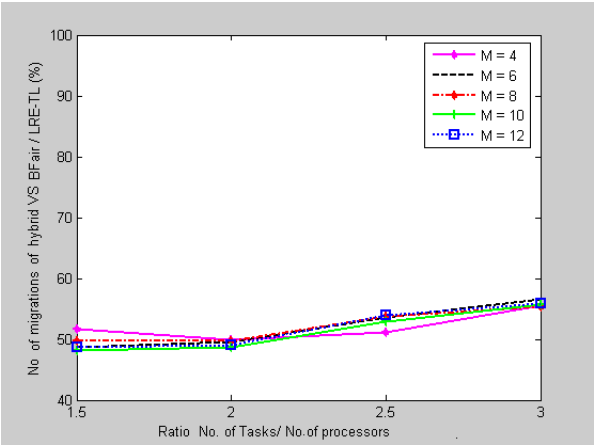

Figure 9. Migration control in variation with number of tasks at U=M

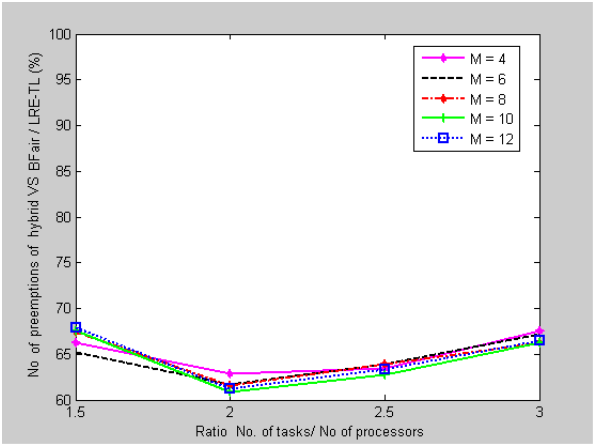

Figure 11. Preemption control in variation with number of tasks at U=M

emption control at $U = 0.75M$ with higher values of task to processor ratio as shown in figure 10. Since this heuristic 2 controls the migration as well, the migration control of hybrid also follows the same pattern specially when $U < M$. At higher value of task to processor ratio, it is difficult for a task to re-execute on the same processor than at lower value of task to processor ratio.

6. Conclusion

In this article, we have shown the improvement in terms of overhead for a class of optimal non-work conserving global scheduling algorithm by using simple heuristics. Our simulation results have validated the efficiency of the heuristics and have shown very clear trends about their properties. The algorithms showed the same trends when we experimented them with four different sets of periods.

At present, we are using the very same heuristics with work-conserving scheduling algorithms. Although work-conserving techniques have intrinsically lower overheads than non-work conserving techniques. We want to evaluate in what proportions such overhead control heuristics may reduce it further.

References

[1] http://storm.rts-software.org.

[2] J. Anderson and A. Srinivasan. Early-release fair scheduling. *In the Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 35–43, 2000.

[3] D. Aoun, A-M Déplanche, and Y. Trinquet. Pfair scheduling improvement to reduce interprocessor migrations. *In the Proceedings of 16th International Conference on Real-Time and Network Systems, Rennes, France*, pages 131–138, 2008.

[4] K. Bletsas B. Andersson and S. K. Baruah. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. *In the Proceedings of the Real-Time Systems Symposium*, pages 385–394, 2008.

[5] A. Block and J. Anderson. Accuracy versus migration overhead in real-time multiprocessor reweighting algorithms. *In the Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 355–364, 2006.

[6] H. Cho, Binoy Ravindran, and E. Douglas Jensen. T-l plane-based real-time scheduling for homogeneous multiprocessors. *Journal of Parallel and Distributed Computing*, 70:225 – 236, 2010.

[7] S. Funk and Vijaykant Nadadur. Lre-tl an optimal multiprocessing scheduling algorithm for sporadic task sets. *In the Proceedings of the 17th International Conferenece of Real-Time and Network systems, Paris*, pages 26–27, 2009.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[9] B. Ravindran H. Cho and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. *In the Proceedings of the IEEE International Real-Time Systems Symposium*, pages 101–110, 2006.

[10] Chiahsun Ho and Shelby H. Funk. A hybrid priority multiprocessor scheduling algorithm. *In the Proceedings of the 31st Real-Time Systems Symposium (RTSS), San Diego, CA, USA*, 2010.

[11] K.S. Hong and J.Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, pages 1326–1331, 1992.

[12] L. George J. Goossens I. Lupu, P. Courbin. Multi-criteria evaluation of partitioning schemes for real-time systems. *In the Proceedings of the 15th IEEE International Conference on Emerging Techonologies and Factory Automation*, 2010.

[13] J.L. Diaz D.F. Garcia J.M. Lopez, M. Garcia. Utilization bounds for multiprocessor rate-monotonic scheduling. *Real-Time Systems*, pages 5–28, 2003.

[14] G. Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. Dp-fair: A simple model for understanding optimal multiprocessor scheduling. *In the Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, pages 3–13, 2010.

[15] R. McNaughton. Scheduling with deadlines and loss functions. *Management Sciences*, pages 1–12, 1959.

[16] Thomas Megel, Renaud Sirdey, and Vincent David. Minimizing task preemptions and migrations in multiprocessor optimal real-time schedules. *In the Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 37–46, 2010.

[17] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Reducing preemptions and migrations in real-time multiprocessor scheduling algorithms by releasing the fairness. In *RTCSA*, 2011.

[18] Paul Regnier, George Lima, Ernesto Massa, Greg Levin, and Scott A. Brandt. Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *RTSS*, pages 104–115, 2011.

[19] N. C.G.Plaxton S.Baruah and D.Varvel. Proportionate progress:a notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[20] M.N Shehzad, A-M Déplanche, Y. Trinquet, and Richard Urunuela. Overhead control in real-time global scheduling. *In the Proceedings of 19th International Conference on Real-Time and Network Systems. Nantes, France*, pages 45–52, 2011.

[21] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *In the Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 189–198, 2001.

[22] R. Stafford. Random vectors with fixed sum. *http://www.mathworks.com/matlabcentral/fileexchange/9700*, 2006.

[23] R. Urunuela, A-M. Déplanche, and Y. Trinquet. Storm - a simulation tool for real-time multiprocessor scheduling evaluation. *In the Proceedings of IEEE International Conference on Emerging Technology and Factory Automation, ETFA, Bilbao*, pages 1–8, 2010.

[24] C. Y. Yang, Jian-Jia Chen, and Tei-Wei Kuo. Preemption control for energy-efficient task scheduling in systems with a dvs processor and non-dvs devices. *In the Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 293–300, 2007.

[25] G. Yao, Giorgio Buttazzo, and Marko Bertogna. Feasibility analysis under fixed priority scheduling with fixed preemption points. *In the Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 71–80, 2010.

[26] D. Zhu, Daniel Mosse, and Rami Melhem. Multiple-resource periodic scheduling problem: How much fairness is necessary? *In the Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 142–151, 2003.